2019 年度 修士論文

ハイブリッド制約処理系 HyLaGI における スケーラビリティ向上に向けた拡張

Enhancement of the Hybrid Constraint Solver HyLaGI for Scalability Improvement

提出日： 2020 年 1 月 29 日
指導： 上田 和紀 教授
研究指導名: 並列知識情報処理研究

早稲田大学 基幹理工学研究科
情報理工・情報通信専攻

学籍番号 : 5118F047-8

佐藤 柾史
Masashi Sato

概要

近年は, 自動運転や IoT システム等の物理世界とコンピュータの相互作用で成り立つシステムの関心が高まっている. そのようなシステムは, 時間進行に伴った内部状態の変化 (連続変化) と, ある条件が成立した際に発生する離散的な状態の変化 (離散変化) の繰り返しで成立する. そのように, 連続変化と離散変化の繰り返しで記述できるシステムをハイブリッドシステムという. 自動運転のように, ふるまいの安全性が最も重要な要件であるハイブリッドシステムの検証は重要である.

ハイブリッドシステムのモデリング言語 HydLa は, 制約プログラミングと制約階層によってハイブリッドシステムを宣言的に記述するための言語である. ハイブリッドシステムの仕様を数式と論理式を用いて簡潔に記述できるという特徴がある. システムの非決定的なふるまいを不等式として記述でき, システムの曖昧さをモデリングできる.

HyLaGI は HydLa 言語を記号実行する処理系であり, 記述された制約を誤差なく解消していくことでハイブリッドシステムの高信頼シミュレーションを行う. 全探索によって非決定的なふるまいを列挙できるため, 曖昧なふるまいをするモデルの安全なシミュレーションと検証を行うことができる.

HyLaGI のシミュレーションにおける困難な点は, その記号計算にある. 多数の数式や複雑な数式が入力された場合は計算コストがかかることが多く, 長いフェーズのシミュレーションが困難だったり求解に失敗したりする. 本研究では, HyLaGI の主要な記号計算手続きに最適化を施し, 計算コストの削減を試みた. 一つは, HyLaGI の各フェーズの制約解消において, 過去に行った計算を再利用することで同じ計算が繰り返されることを抑制し, 長いフェーズのシミュレーションに耐えうるように最適化を施した. もう一つは, 離散変化原因が多数ある場合に HyLaGI はその全てに対して実際に離散変化が発生するかどうかの最小化問題を解くため, 実行のボトルネックとなってしまう問題の解決を試みた. 離散変化原因の探索に分枝限定法を導入し, 不要な解探索を行わないようにするに最適化を施した.

結果, 前者の問題は不要な再計算が抑制され, 特に制約解消のコストが大きい問題の実行時間を 30-50% 短縮することができた. 後者の問題では, モチベーションとなる問題のボトルネックを解消することができ, 離散変化の候補となる制約の数を n とすると, 最小化問題を解く回数が O(n) から O(1) に削減することができた.

また, HyLaGI に存在量化子を導入し, 制約を compositional に結合してプログラムを組みやすくなるように拡張した.

Abstract

In recent years, there has been an increasing interest in systems such as autonomous driving systems and IoT systems in which the physical world and computers interact with each other. These systems alternate continuous state changes over time and discrete changes that are caused when certain conditions are satisfied. Systems which can be represented by discrete changes and continuous changes are called hybrid systems. For hybrid systems such as autonomous driving systems in which the safety of behavior is a key requirement, verification of the systems are very important

HydLa is a declarative language for modeling hybrid systems. HydLa adopts the constraint programming paradigm and constraint hierarchies. Thanks to these features, we can concisely describe properties of hybrid systems using mathematical and logical formulas. In particular, they allow us to model uncertain behaviors of hybrid systems using inequalities.

HyLaGI is an implementation of HydLa to simulate HydLa programs symbolically. HyLaGI allows us to rigorously calculate all trajectories represented by inequalities by constraint satisfaction, which realizes safe simulation and verification of uncertain hybrid systems.

Symbolic simulation of HydLa programs based on constraint solving is quite challenging. Simulation of HydLa programs that contain a lot of constraints or complex formulas may incur high calculation cost or result in unsuccessful simulation. In this research, we apply two optimization techniques to the main symbolic procedure of HyLaGI. One is that we reuse the results of previous calculations. The other is that we reduce the calculation cost of exhaustive search to find the time of the next discrete changes. For this purpose, we introduce a branch-and-bound algorithm to the finding of the causes of discrete changes to reduce the search space.

The former optimization technique reduced the calculation cost by 30-50 %. The latter optimization technique reduced the time complexity of minimization problems of the motivating example from O(n) to O(1).

In addition, we introduced an existential quantifier to HyLaGI, which enabled us to write programs by combining constraints compositionally.

目次

図目次

第 1 章

はじめに

1.1　研究の背景と目的

近年は, 自動運転や IoT システム等の物理世界とコンピュータの相互作用で成り立つシステムへの関心が高まっている. そのようなシステムは, 時間進行に伴った内部状態の変化 (連続変化) と, ある条件が成立した際に発生する離散的な状態の変化 (離散変化) の繰り返しで成立する. その様に, 連続変化と離散変化の繰り返しで記述できるシステムをハイブリッドシステム [1] という. 自動運転のように, ふるまいの安全性が最も重要な要件であるハイブリッドシステムの検証は重要である.

我々は, ハイブリッドシステムを制約プログラミングのパラダイムで記述するための言語 HydLa [3] を開発している. HydLa は論理式と数式, 及び制約階層のみでハイブリッドシステムを記述するように設計された言語であり, プログラミング言語として新しい記法を学習することをほとんど要求せず, 数学者や物理学者を対象に開発されている. HyLaGI [5] は記号計算による高信頼なシミュレーションを実現する HydLa の実行系で, 不等式による非決定性をはらむ HydLa プログラムを全探索によって解軌道を全列挙するのが特徴である.

記号計算によるシミュレーションは数値計算と違って丸め誤差がなく, 定性的な振る舞いを間違えないという利点があるが, HyLaGI のような不等式の入った制約解決を行うには強力なソルバが必要であり, 実行コストが高く付くという欠点がある. HyLaGI のバックエンドソルバは Mathematica[8] であり, HydLa プログラムを実行するに足る能力を持っているが, 解析の難しい数式や複数の不等式が出現するプログラムは依然として求解が困難であり, 実行コストがかかったりシミュレーションが止まったりしてしまうという問題がある.

本研究は HyLaGI のスケーラビリティの向上を目的とし, 主要な記号計算処理に最適化を施した. 一つは, HyLaGI の各フェーズの制約解消において, 過去に行った計算を再利用することで同じ計算が繰り返されることを抑制し, 長いフェーズのシミュレーションに耐えうるように最適化を施した. もう一つは, 離散変化原因が多数ある場合に HyLaGI はその全てに対して実際に離散変化が発生するかどうかの最小化問題を解くため, 実行のボトルネックとなってしまう問題の解決を試みた. 具体的には, 離散変化原因の探索に分枝限定法を導入し, 不要な解探索を行わないようにするに最適化を施した.

また, HyLaGI に存在量化子を導入し, 制約を compositional に結合してプログラムを組みやすくなるように拡張した.

1.2　論文構成

第 2 章では, 研究の背景にあるハイブリッドシステムとプログラミング言語 HydLa, 及び処理系 HyLaGI についての説明を行う. 第 3 章では, HydLa, HyLaGI への存在量化子の導入について, その意味や構文を例題を用いて説明する. 第 4 章では, 計算再利用による最適化について, 背景となる技術と実行アルゴリズムを提示し, 例題による評価を行う. 第 5 章では, 離散変化時刻の導出手続きの最適化について, 分枝限定法の説明と実装アルゴリズムを説明し, 例題による評価を行う. 第 6 章では関連研究について, 第 7 章ではまとめと今後の課題について述べる.

第 2 章

プログラミング言語 HydLa とその処理系 HyLaGI

本章では, ハイブリッドシステムと, ハイブリッドシステムモデリング言語 HydLa, および, HydLa の処理系 HyLaGI について説明する.

2.1 ハイブリッドシステム

ハイブリッドシステム [1] とは, 連続変化と離散変化を繰り返す動的システムであり, 物理世界とコンピュータ (サイバー世界) の相互作用で成り立つシステム (CPS) 等が挙げられる。例として, 物理システムでは床を跳ねるボールがハイブリッドシステムであり, 重力に従って落下している際は連続変化で, 床に当たって速度が反転する際が離散変化である. 他にも, 制御システムは多くがハイブリッドシステムとしてモデリング可能である.

2.2 HydLa について

HydLa [3] はハイブリッドシステムをモデリングする言語である. HydLa の特徴として以下が挙げられる.

- ハイブリッドシステムの構成要素 (内部変数) を制約 (数式と論理式) を用いて記述する宣言型言語である
- 不等式制約を用いることで, ふるまいの非決定性を表現できる
- 制約階層 [2] を用いて制約に優先度を定義することで, 制約の矛盾を解決する

2.2.1　HydLa プログラムの例

図 2.1 に床を跳ねる質点の HydLa プログラムを示す.

```
1  INIT <=> y = 10 & y' = 0.
2  FALL <=> [](y'' = -10).
3  BOUNCE <=> [](y- = 0 => y' = -4/5 * y'-).
4
5  INIT, FALL << BOUNCE.
```

図 2.1　床を跳ねる質点の HydLa プログラム

HydLa プログラムは, 大きく制約モジュールの定義と制約モジュールの宣言に分けられる. 1-3 行目は制約モジュールの定義である. HydLa プログラムでは, 制約を再利用可能なように別名をつける. 引数を設けることが可能であり, 変数や値を入力できる.

INIT は質点に対応する変数 y を初期化するモジュールであり, 質点の初期位置が 10 で速度が 0 であることを述べている. なお, HydLa の変数は時刻の関数であり, y' は $\frac{dy}{dt}$ を表す。FALL は質点の加速度が-10 であることを述べている. 記号"[]"は、LTL 論理式の □ 作用素である。□ は, 修飾されている制約が t>0 で常に成り立つことを要求する. 逆に, 上述の INIT モジュール内の制約の様に, □ のついていない制約は時刻 0 のみで成り立つ制約である. BOUNCE は, 概念的にはボールが床に衝突した時に跳ねることを記述したモジュールである. 論理演算子 ⇒ を用いた制約は条件付き制約と呼ばれ, 前件 (ガード条件) が成立している時刻で後件の制約が有効であることを要求する. また, 後置演算子-が付いた変数 (prev 変数) は, 制約が成立している時刻の左近傍の変数値を参照している. BOUNCE の例では, y-=0 が成立した時刻で y'=-4/5y'-が成立しており, y-は離散変化前の質点の速度を参照している.

5 行目は制約の宣言である. 定義した制約に優先度を設けて宣言する. ","演算子は右辺と左辺の制約に優先度関係が無いことを宣言するものであり, この例では INIT と FALL は並列の関係である. "≪"演算子は, 右辺が左辺よりも優先度が高いことを宣言するものであり, この例では FALL よりも BOUNCE の方が高い優先度を持つ.

優先度関係は, 制約同士が矛盾した際の制約の解決で用いられる. HydLa プログラムは

宣言された制約をできるだけ多く採用しようとするが, 矛盾した際には優先度の低い制約を取り除いて再度採用を試みる. 厳密には, 以下の式を満たす無矛盾かつ要素数が最大のモジュール集合 (極大無矛盾集合) を採用する.

$$\forall M_1, M_2((M_1 \ll M_2 \wedge M_1 \in MS) \Rightarrow M_2 \in MS) \tag{2.1}$$

$$\forall M(\neg\exists(R \ll M) \Rightarrow R \in MS) \tag{2.2}$$

式 2.1 は, 優先度の低いモジュールが採用される場合は優先度の高い制約も同様に採用されていることを要請し, 式 2.2 は自分より優先度の高いモジュールが存在しないモジュール (required なモジュール) は必ず採用されることを要請する. 厳密な意味論は文献 [3] を参照してほしい. この例では, `y-=0` が成立した時刻で, `FALL` と `BOUNCE` が矛盾する. その際, 優先度の低い `FALL` を取り除くことで, `BOUNCE` を採用することができる. この考えにより, 基本的には常に成り立っているが, ある条件下で成り立たなくなる挙動をモデリングするのに便利である. 多くの記述例では, 条件付き制約を高い優先度に置くことで離散変化の時に成り立つ条件を記述する.

2.3　HydLa の処理系 HyLaGI

HyLaGI [5] は HydLa 言語の実行系である. HydLa 言語は, HydLa の意味論に従った解軌道をどのように計算するかについて言及していない. 処理系 HyLaGI は以下の特徴を持つ

- 記号計算により, 丸め誤差を許容しないシミュレーションを行う
- 記号計算が難しい問題では, 区間値を用いた精度保証数値計算を用いることができる
- 非決定的なふるまいは, 全探索により全ての解軌道を導出できる

記号計算は, ハイブリッドシステムの安全な検証に必要な要件であり, 丸め誤差によって本来あるべきふるまいを損なうことを防ぐ. バックエンドの記号ソルバとして Mathematica を備えており, それ以外の手続きは C++ で実装されている.

2.3.1　HyLaGI の実行アルゴリズム

図 2.2 に HyLaGI 全体の実行アルゴリズムを示す. なお, 本小節で用いられる実行アルゴリズムは文献 [5] からの引用である.

HyLaGI はまず, 入力された HydLa プログラムから式 2.1, 2.2 に基づいて極大無矛盾集

合になりうるモジュールの集合 (解候補モジュール集合) を導出し, 要素数でトポロジカルソートした状態で保持する (行 1). 7 行目から 23 行目のループでは, 解候補モジュール集合から各時刻の解軌道を求める処理である. 離散変化時の変数の値を求めるポイントフェーズ (PP) と, 連続変化時の微分方程式を求めるインターバルフェーズ (IP) を繰り返して解軌道を求める.

PP, IP では共通して, そのフェーズの極大無矛盾集合を求める手続き `MCS` がある (行 10). `MCS` の具体的な処理内容を図 2.3 に示す. `MCS` は, 変数の左近傍値と変数が参照する記号パラメータ, 解候補モジュール集合から, 極大無矛盾集合とその極大無矛盾集合になるための記号パラメータを計算する関数である. 記号パラメータは, 変数の値が不等式で表される不定値である場合に導入されるパラメータであり, 変数に参照される. 不等式の取る値に依って極大無矛盾集合が変わる可能性があるため, この関数は非決定性を持つ. 時刻 0 では □ 制約を取り除く処理があるが, 基本的には解候補モジュール集合を走査して変数の初期値との充足性の判定をする処理 (行 6 の `CalculateClosure`) を行う. `CalculateClosure` の詳細は文献 [5] を参照してほしいが, ガード条件の成立有無の判定と展開された制約同士が無矛盾であるかの制約の充足性判定を収束するまで行う. PP と IP では異なる `CheckConsistency` 関数が入力される. PP の `CheckConsistency` 関数は, 変数の値を連立方程式を解くことで求め, IP では微分方程式を解くことで求める点で異なる. その他に, 引数の `E` はシミュレーションの過程で展開された □ 制約である. HydLa では条件付き制約の後件に □ 制約を記述することを許しており, ガード条件が成立した時刻から後件の □ 制約がモジュール内で展開され, 有効になる. 極大無矛盾集合の導出において後件の □ 制約は, 変数の左近傍値以外で唯一求解結果に因果的であり, 特殊な扱いをしている.

IP では, 次の離散変化の発生時刻を求める処理がある (行 21). HydLa の離散変化は, ガード条件の成立の有無が変更された際に発生する. `MCS` において, 微分方程式を解くことで導出した時刻の式と, ガード条件の元で時刻を最小化する問題を解き (`FindMinTime` 関数), それらの中で最小のものを求める (`CompareMinTime` 関数). この処理も, 記号パラメータに依ってどのガード条件が最初に成立の有無が変更されるかが変わるため, 非決定性がある.

Require: *HydLa*: basic HydLa program,
MaxT: maximum simulation time

1: $MS := TopologicalSort(HydLa)$ // list of candidate sets of constraints
2: $V := GetVariables(HydLa)$
3: $T := 0$ // current time
4: $S := true$ // current constraint store
5: $P := true$ // constraints on symbolic parameters
6: $E := \emptyset$ // expanded consequents
7: **while** $T < MaxT$ **do**
8: // Point Phase (PP)
9: $S := Subst(S, T)$
10: $(S, P, E, \_, \_) := MCS(S, MS, E, P, T, CheckConsistencyPP)$
11: **if** $S = false$ **then**
12: break
13: **end if**
14: $(S, P) := AddParameters(S, P, V)$
15: // Interval Phase (IP)
16: $(S, P, E, A_-, A_+) := MCS(S, MS, E, P, T, CheckConsistencyIP)$
17: $S := SolveDifferentialEquation(S)$
18: **if** $S = false$ **then**
19: break
20: **end if**
21: $(MinT, P) := GetElement(CompareMinTime($
$(\bigcup_{(g \Rightarrow c) \in A_-} FindMinTime(Subst(g, S), P))$
$\cup (\bigcup_{(g \Rightarrow c) \in A_+} FindMinTime(Subst(\neg g, S), P))$
$\cup \{(MaxT - T, true)\}))$
22: $T := MinT + T$
23: **end while**

図 2.2　HyLaGI の実行アルゴリズム (文献 [5] より引用)

Require: S: constraint store, MS: list of candidate constraint sets,
E: set of expanded always consequents,
P: constraint on symbolic parameters,
T: current time, *CheckConsistency*: function for consistency checking
Ensure: constraint store, constraint on symbolic parameters, set of expanded always, maximal consistent set, set of not entailed guards, set of entailed guards

1: **for** $M \in MS$ **do**
2: **if** $T > 0$ **then**
3: $M := EliminateNotAlways(M)$
4: **end if**
5: $(S_{tmp}, E_{tmp}, P, A_-, A_+) :=$
6: $CalculateClosure(S, M, P, E, CheckConsistency)$
7: **if** $S_{tmp} \neq false$ **then**
8: **return** $(S_{tmp}, P, E_{tmp}, M, A_-, A_+)$
9: **end if**
10: **end for**
11: **return** $(false, P, E, \emptyset, \emptyset, \emptyset)$

図 2.3　MCS の実行アルゴリズム (文献 [5] より引用)

第 3 章

存在量化子を用いた変数の動的な生成

HydLa でプログラムで使われた変数は静的なものであり, あらゆる文脈からも参照され, 各時刻において付値を持つ. モジュール記法を用いて制約のインスタンスを生成する場合や, 一時的な変数を生成して制約の伝搬を行う場合において, 動的な変数は便利である. 文献 [3] で存在量化子を用いた変数の動的な生成手法が提案されている. 本章では, その手法の説明を行い, その構文・意味の説明, HyLaGI への実装と例題への適用結果について述べる.

3.1 HydLa における存在量化子

HydLa の制約の意味は一般的に用いられる論理式ですべて説明されるため, 論理記号を用いた構文・意味の拡張は自然である. 存在量化子を用いることで, スコープの外から参照されない変数を動的に生成することができる.

3.1.1 存在量化子記法を組み込んだ構文

導入した構文を図 3.1 に示す.

存在量化子は HydLa の構文における制約を修飾することができ, 数式に含まれる明示した変数を束縛する.

(hydla program)	P	::=	(DF | DC)*
(definition)	DF	::=	dname($\vec{X}$){DC} | cname($\vec{X}$) ⇔ C
(constraint)	C	::=	A | C ∧ C | G ⇒C | □C | ∃$vname$. C | cname($\vec{E}$)
(guard)	G	::=	A | G ∧ G | G ∨ G
(atomic constraint)	A	::=	E RO E
(relational operator)	RO	::=	= | ≠ | < | ≤ | > | ≥
(expression)	E	::=	E AO E | P | $constant$
			| $unary\_function(E)$
(arithmetic operator)	AO	::=	+ | − | mes | ÷ | ^
(previous)	P	::=	D | $D-$
(derivative)	D	::=	$vname$ | $vname'$
(declaration)	DC	::=	M | DC, DC | DC ≪ DC
			| dname($\vec{E}$)
(module)	M	::=	C

図 3.1　存在量化子記法を導入した HydLa の構文 (文献 [5] から引用・追加)

3.2　存在量化子の意味

HydLa プログラムに明示された存在量化子は, 制約が展開された時刻で stolem 化により除去される. 量化された変数は, プログラムに出現しない変数名が新しく割り当てられ, リネームされる. HydLa の意味論 (文献 [5] を参照) より, 条件付き制約の後件の制約は前件が成立した時刻で展開される. 従って, 後件に出現した存在量化子は, 前件が成立した時刻で除去される.

構文より, 存在量化子は制約のどの部分に組み込んでもよいこととなっている. 意味は論理式に従うが, 時相論理式と存在量化子を組み合わせると実装, シミュレーションが困難な場合がある. 具体的には, □ 作用素の内側に存在量化子があり, かつその制約が連続変化フェーズで有効な場合である. 図 3.2 に簡単な例を示す.

変数 f は 0 から単調増加し, モジュール B の前件が $t = [1, 2]$ の区間で成立する. その区間内の各時刻で変数が生成されるため, 不可算無限個の変数が生成されることになり, 実装は不可能である. 明らかにユーザーの本位とする挙動ではないため, 処理系はエラーを生成して実行を中断するのが望ましい.

```
1 A <=> f=0 & [](f'=1).
2 B <=> [](1<=f<=2 => \x.([]x=1)).
3
4 A, B.
```

図 3.2　連続変化フェーズで成立する always 制約の内側にある存在量化子の例

3.3　存在量化子の利用

本節では, HydLa プログラムにおける存在量化子が適用可能な例を紹介する

3.3.1　モジュール内の局所変数

HydLa には変数に別名をつけ, 引数として変数や値を注入できるモジュール記法があるが, HydLa の変数はスコープがグローバルであるため, 内部的な変数が外部に漏れ出す. そこで, モジュール内で使われる局所変数を存在量化子で除去することで, ローカル変数を実現することができる. 存在量化子を用いた局所変数の例として, 図 3.3 にスイッチを用いた RC 直列回路の HydLa プログラムを示す.

プログラムでは, アトミックな回路素子を並列合成してスイッチのある RC 回路を組み立てている. その際, 回路素子の内部変数を存在量化子で隠蔽し, 外部から補足できないようにしている. なお, プログラム内では, 存在量化子は記号"\"で表される制約 C はコンデンサに対応する制約であるが, 内部で電荷に関する微分方程式を解き, 出力の変数に伝搬している. q を内部変数にすることで, C を再利用した際に変数の衝突を防ぐことができる. 存在量化子を使わない場合は, コンデンサの数だけ別々の変数を用意し, 引数として C に注入する必要がある.

3.3.2　制約生成の遅延

制約の発行時に一時変数を用いて計算をする一つの例として, 制約生成の遅延がある. 条件の成立をトリガーとしてイベントを発生させるモデルでは, 条件の成立からイベントの発生までに応答遅延があるのが現実的である. 例として, 遅延を含んだサーモスタットの HydLa プログラムを図 3.4 に示す. 変数 p は温度を表し, モジュール ON, OFF によって

```
1  R(vin,vout,i,r) <=> [](vin-vout = r*i).
2  C(vin,vout,i,c) <=> \q.(q=0 & [](q=c*(vin-vout) & i=q')).
3  E(e,e0) <=> [](e=e0).
4  TIMER(timer) <=> timer=0 & []timer'=1.
5  SWITCH(vin,on,vout) <=>
6    [](on=0 => vout=0) & [](on=1 => vin=vout).
7
8  STEP(time,on) <=> \timer.(TIMER(timer) & [](timer<time => on=0)
9    & [](timer>=time => on=1)).
10 TIMER_SWITCH(vin,vout,time) <=>
11   \on.(STEP(time,on) & SWITCH(vin,on,vout)).
12
13 RC(vin,vout,i,r,c,vc) <=>
14  \vtmp.(R(vin,vtmp,i,r) & C(vtmp,vout,i,c) & [](vc=vtmp-vout)).
15
16 E(e,5), TIMER_SWITCH(e,vin,1), RC(vin,0,i,100,0.1,vc).
```

図 3.3　存在量化子記法を用いたスイッチ付き RC 直列回路の HydLa プログラム

単調増加, 単調減少する. モジュール SWITCHON, SWITCHOFF は, 温度が閾値 (62, あるいは 68) になった際に成立し, 変数 p の微分方程式を変更する. SWITCHON, SWITCHOFF が成立するたびに存在量化子によってローカルタイマーを生成する. タイマーは [0.1, 0.5] で初期化され, 単調減少し, 値が 0 になった時刻で温度 p の微分方程式を変更する制約を発行する.

このプログラムを HyLaGI で実行した際の出力を図 3.6 に, 解軌道を図 3.5 に示す.

時刻 p が閾値に到達する度にタイマーが区間で初期化され, それが微分方程式の変更時刻に伝搬されるため, 各離散変化でパラメータが導入される.

HyLaGI においては, 存在量化子が除去される時刻で新しく生成される変数に機械的に名前が割り当てられる. 具体的には, 束縛変数名に連番と生成されたフェーズ番号が付加される.

```
1  INIT <=> p = 65 & mode = 0.
2  OFF <=> [](mode = 0 => p' = -2).
3  ON <=> [](mode = 1 => p' = 1).
4  MODE(l,r,m) <=> \x.(l < x < r & [](x' = -1)
5                    & [](x- = 0 => mode = m)).
6  SWITCHOFF <=> [](p- = 68 & mode = 1 => MODE(0.1,0.5,0)).
7  SWITCHON <=> [](p- = 62 & mode = 0 => MODE(0.1,0.5,1)).
8
9  INIT, [](mode' = 0) << (SWITCHON, SWITCHOFF).
10 OFF, ON.
11
12
13 //#hylagi -p8
```

図 3.4　遅延のあるサーモスタットの HydLa プログラム

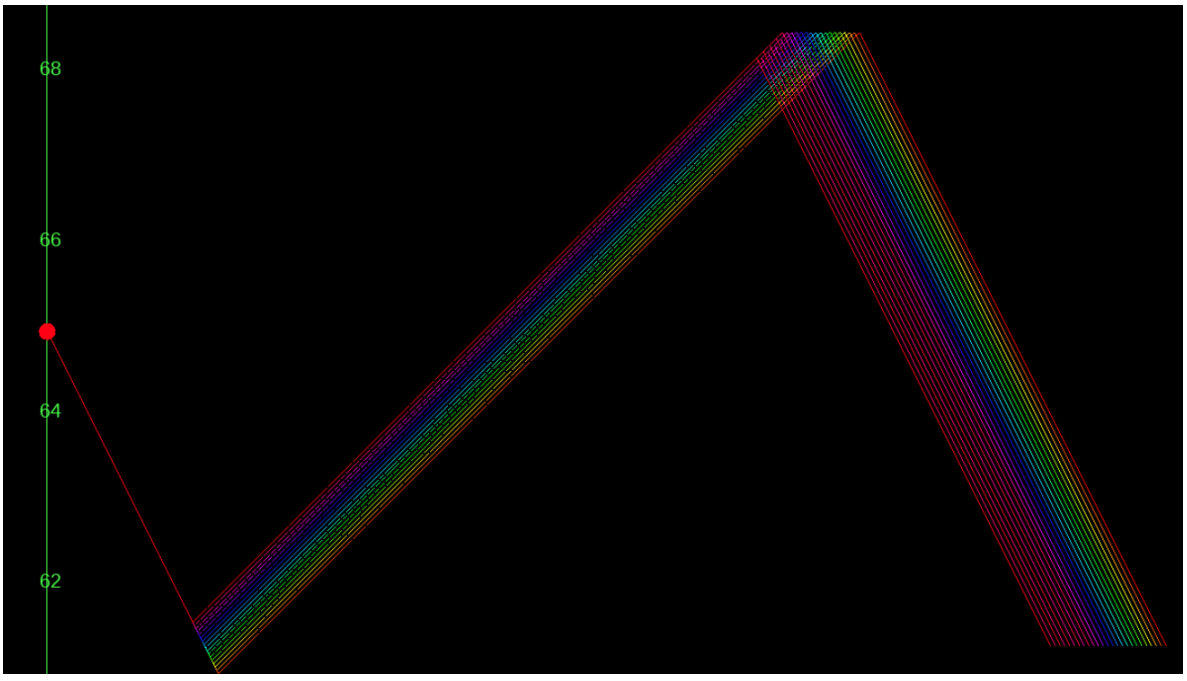

図 3.5　遅延のあるサーモスタットの解軌道

```
---------Case 1---------
---------1---------
---------PP 1---------
unadopted modules: {}
positive  : mode=0 => p'=-2
negative  :
t : 0
mode : 0
p : 65
mode': 0
p': -2
---------IP 2---------
unadopted modules: {}
positive  :
negative  :
t : 0->3/2
mode : 0
p : 65+t*(-2)
mode': 0
p': -2
---------2---------
---------PP 3---------
unadopted modules: {}
positive  : p-=62&mode=0
    => MODE(02/10, 05/10, 1)
negative  :
t : 3/2
mode : 0
p : 62
x1$3 : p[x1$3, 0, 3]
mode': 0
p': -2
x1$3': -1
---------IP 4---------
unadopted modules: {}
positive  :
negative  : p-=62&mode=0
    => MODE(02/10, 05/10, 1)
t : 3/2->p[x1$3, 0, 3]+3/2
mode : 0
p : 65+t*(-2)
x1$3 : t*(-1)+p[x1$3, 0, 3]+3/2
mode': 0
p': -2
x1$3': -1
```

```
---------3---------
---------PP 5---------
unadopted modules: {mode'=0}
unsat modules : {SWITCHON, mode'=0}
unsat constraints : {mode'=0, mode=1}
positive  : mode=1 => p'=1
  x1$3-=0 => mode=1
negative  : mode=0= >p'=-2
t : p[x1$3, 0, 3]+3/2
mode : 1
p : 62+(-2)*p[x1$3, 0, 3]
x1$3 : 0
p': 1
x1$3': -1
---------IP 6---------
unadopted modules: {}
positive  :
negative  : x1$3-=0
      => mode=1
t : p[x1$3, 0, 3]+3/2->3*p[x1$3, 0, 3]+15/2
mode : 1
p : t+(-3)*p[x1$3, 0, 3]+121/2
x1$3 : t*(-1)+p[x1$3, 0, 3]+3/2
mode': 0
p': 1
x1$3': -1
---------4---------
---------PP 7---------
unadopted modules: {}
positive  : p-=68&mode=1
      =>MODE(02/10, 05/10, 0)
negative  :
t : 3*p[x1$3, 0, 3]+15/2
mode : 1
p : 68
x1$3 : -2*(3+p[x1$3, 0, 3])
x2$7 : p[x2$7, 0, 7]
mode': 0
p': 1
x1$3': -1
x2$7': -1
---------IP 8---------
unadopted modules: {}
positive  :
negative  : p-=68&mode=1
      =>MODE(02/10, 05/10, 0)
t : 3*p[x1$3, 0, 3]+15/2
      ->3*p[x1$3, 0, 3]+p[x2$7, 0, 7]+15/2
mode : 1
p : t+(-3)*p[x1$3, 0, 3]+121/2
x1$3 : t*(-1)+p[x1$3, 0, 3]+3/2
x2$7 : t*(-1)+3*p[x1$3, 0, 3]+p[x2$7, 0, 7]+15/2
mode': 0
p': 1
x1$3': -1
x2$7': -1
```

図 3.6　遅延のあるサーモスタットの出力

3.3.3　制約の再帰的な定義

存在量化子を用いることで, 制約を再帰的に展開することができる. 再帰的な計算を行うためには, 呼び出し元へ計算を伝搬させるための中間変数が必要である. 例として, 離散変化フェーズで階乗を計算し, 微分方程式の初期値として伝搬させる例を図 3.7 に示す

```
1 FACTORIAL(ans, n)
2         <=> (n = 0 => ans = 1)
3           & (n > 0 => \x.(ans = n * x & FACTORIAL(x, n-1))).
4 CALC_F <=> [](timer- = 1 => FACTORIAL(x, 5) & [](x'=1)).
5 TIMER <=> timer = 0 & [](timer' = 1).
6
7 TIMER, CALC_F.
```

図 3.7　離散変化フェーズに 5 の階乗を計算するプログラム

FACTORIAL モジュールは自身を再帰的に呼び出して階乗を計算し, 変数 ans に計算結果を伝搬させる. FACTORIAL が展開されるたびに存在量化子を除去し, 中間変数を導入する. HyLaGI の出力を図 3.8 に示す.

出力より, PP 3 において展開された FACTORIAL の条件付き制約が同じ時刻で一斉に発火し, 各階乗の値を生成された変数を通して伝搬させていることが分かる. HydLa の変数は時刻の関数であり, 各時刻で複数の値を保持してマルチステップの計算を行う superdense time を採用していない. 存在量化子による動的な変数を用いることで, マルチステップの計算を時刻を進行させずに行うことができ superdense time の代替になりうる.

```
------ Result of Simulation ------
---------Case 1---------
---------1---------
---------PP 1---------
unadopted modules: {}
positive  :
negative  :
t : 0
timer : 0
timer': 1
---------IP 2---------
unadopted modules: {}
positive  :
negative  :
t : 0->1
timer : t
timer': 1
---------2---------
---------PP 3---------
unadopted modules: {}
positive  : timer-=1 =>FACTORIAL(x, 5)&[](x'=1)
  5>0 =>\x1$.x=5*x1$3&FACTORIAL(x1$, 5-1)
  5-1>0 =>\x2$.x1$3=(5-1)*x2$3&FACTORIAL(x2$, 5-1-1)
  5-1-1>0 =>\x3$.x2$3=(5-1-1)*x3$3&FACTORIAL(x3$, 5-1-1-1)
  5-1-1-1>0 =>\x4$.x3$3=(5-1-1-1)*x4$3&FACTORIAL(x4$, 5-1-1-1-1)
  5-1-1-1-1>0 =>\x5$.x4$3=(5-1-1-1-1)*x5$3&FACTORIAL(x5$, 5-1-1-1-1-1)
  5-1-1-1-1-1=0 =>x5$3=1
negative  :
t : 1
timer : 1
x : 120
x1$3 : 24
x2$3 : 6
x3$3 : 2
x4$3 : 1
x5$3 : 1
timer': 1
x': 1
---------IP 4---------
unadopted modules: {}
positive  :
negative  : timer-=1 =>FACTORIAL(x, 5)&[](x'=1)
t : 1->Infinity
timer : t
x : 119+t
x5$3 : 1
timer': 1
x': 1
# time reached limit
```

図 3.8　階乗計算プログラムの出力結果

第 4 章

計算再利用による最適化

HydLa におけるシミュレーションやシミュレーションによる性質の検証においては, 基本的には各フェーズで毎回軌道の計算を行うことで実現している. HydLa で記述対象のハイブリッドシステムのモデルの多くは, 同じ極大無矛盾集合を求めることを各フェーズ繰り返すことが多く, HyLaGI はそのようなモデルに対して同じ数式処理を繰り返し実行する. 本研究では, 制約解決の手続きを動的に解析し, 同じ計算を繰り返さないように入出力の対応関係を保存しておく最適化を提案し, 処理系に組み込んだ.

4.1 最適化対象の処理について

4.1.1 極大無矛盾集合の導出

HyLaGI の実行処理における要は, 各フェーズに行われるモジュールの極大無矛盾集合の導出処理である. 図 2.3 が具体的な処理である. `MSC` 関数では, 極大無矛盾集合の候補 (解候補モジュール集合) をイテレートし, 極大無矛盾集合になりうるかを判定する `CalculateClosure` 関数を呼び出す. `CalculateClosure` 関数では, 前フェーズの変数への付値と極大無矛盾集合の候補を評価し, 制約が充足可能であるかを判定する. 多くの問題では, 極大無矛盾集合の導出における制約解消のための数式処理が実行のボトルネックとなる. その一例として, 5 つの水槽の水量調節をモデルした HydLa プログラムを挙げる. プログラムを図 4.10 に, 実行時間のグラフを図 4.2 に示す.

極大無矛盾集合導出処理が実行時間の 8 割以上を占めている. 関数 `MSC`, `CalculateClosure` における最適化は文献 [7] 等で述べられているように, 極大無矛盾集合の動的な導出や変数の依存関係を用いた手法が実装されているが, 実質的に必要

```
1  Xs := {x1..x5}.
2
3  INIT(x,xinit,vinit) <=> x = xinit /\ x' = vinit.
4  CONST <=> [](inflow = |Xs|).
5  LINEAR(x) <=> [](x'' = 0).
6  INFLOW(x1,r1, outflow)
7       <=> [](x1- = r1- => x1' = inflow - outflow).
8  NO_INFLOW(x1,r1,x2, outflow) <=>
9       [](x1- = r1- & x2'- = inflow - outflow => x2' = -outflow).
10
11 INIT(Xs[1], 1, inflow - 1), CONST,
12 { INIT(Xs[i], 2 + i/2, -1-((i-1)/17) ) | i in {2..|Xs|} }.
13 { LINEAR(Xs[i]) << INFLOW(Xs[i], 1, 1+((i-1)/17) )
14                                    | i in {1..|Xs|} }.
15 { LINEAR(Xs[j]) << NO_INFLOW(Xs[i], 1, Xs[j], 1+((j-1)/17))
16                       | i in {1..|Xs|}, j in {1..|Xs|}, i != j }.
```

図 4.1　5 つの水槽の水量調節プログラム

な計算は削減することはできない. 本研究の最適化手法は, HyLaGI が過去のフェーズで行った計算を繰り返している点に焦点を当て, 一度行った計算結果を再利用するものである.

4.1.2　極大無矛盾集合導出手続きの性質

MSC は純粋な関数であり, 入力が同じなら出力も同じである. 図 2.3 を参考に, 関数の入力について説明する. まず, 入力 MS は解候補モジュール集合であり, シミュレーションにおいて不変である. 関数の手続きを解析すると, 入力 T は時刻 0 以外なら実行結果に関与しないことが分かる. 入力 E は過去に展開された条件付き制約の後件にある □ 制約である. 入力 CheckConsistency は, PP, IP に対応した高階関数である. 簡単のため, $t > 0$ かつ後件の □ 制約が展開されないことを仮定すると, 関数 MCS の実行結果は前フェーズの制約である S と記号パラメタ P, および, フェーズが PP であるかどうかによって決まる. MCS の内部で呼び出される CalculateClosure において, 入力された制約 S は評価中の解候

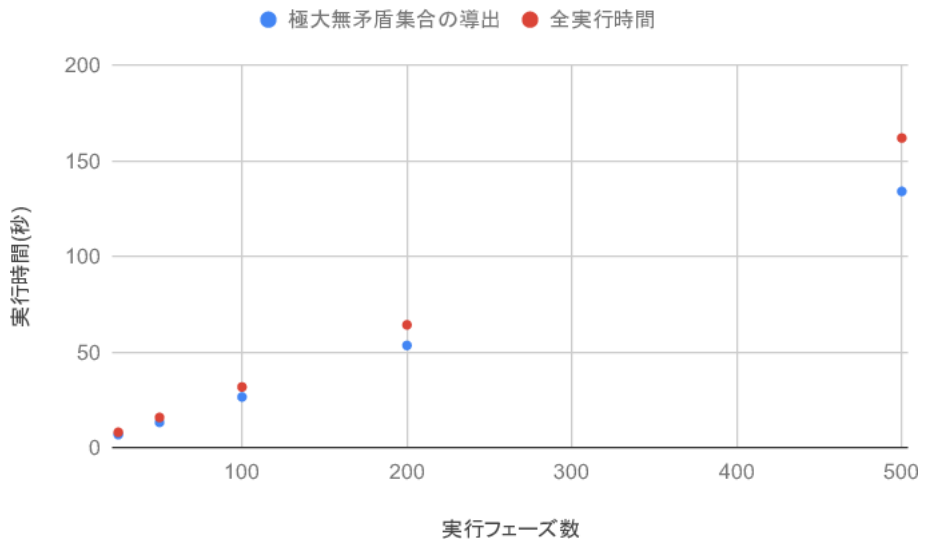

図 4.2　5 つの水槽の水量調節プログラムの実行フェーズと実行時間の関係

補モジュール集合に左極限値として明示的に参照されるほか, 左連続である変数によって暗黙的に参照される. また, 記号パラメタ P は入力された制約 S によって参照される. 出力は変数に割り当てられる式であり, 採用されたモジュールの集合と成立した条件付き制約の組と言い換えることもできる. 本研究では, 各 PP, IP において, 関数 MCS の入出力の対応関係の組を保存し, 可能ならば MCS の処理を短絡して出力を取り出すことで最適化をはかる. また, HydLa の宣言的意味論 (文献 [3]) より, ある時刻の変数の値は直前の時刻の変数の値と展開された ロ 制約においてのみ因果的であることがわかり, 本小節で説明した MCS 関数の入力および動作は HydLa において本質的なものであることを注記する.

4.1.3　極大無矛盾集合導出手続きの動作

極大無矛盾集合導出手続きでは, 前フェーズの変数値と解候補モジュール集合を用いて以下の判定を収束するまで行う.

1. ガード条件の成立有無
2. 採用候補の制約同士の無矛盾性

図 4.3 に示す例題を用いて, PP における関数 MCS の動作を確認する.

最初のフェーズ以外のポイントフェーズにおいて, MCS への入力 x-の値に応じて 3 つのケースに分岐し得る.

```
1 A <=> [](y=1).
2 B <=> [](0<=x-<=2 => y=x-).
3 A << B.
```

図 4.3　MCS の動作確認例題

入力	出力
$0 \leq x- < 1 \vee 1 < x- \leq 2$	$y = x-$
$x- = 1$	$y = x- \wedge y = 1$
$x- < 0 \vee x- > 2$	$y = 1$

表 4.1　プログラム 4.3 における, 極大無矛盾集合導出手続きの入出力対応関係

1. $0 \leq x- < 1 \vee 1 < x- \leq 2$ の場合
   変数 x-はモジュール B の前件と後件によって明示的に参照され, 無矛盾性判定に用いられる. この場合, モジュール B の前件が成立し, 後件が発行されることによって A と B が矛盾する. 結果的に, B のみが採用され, 後件の y=x-が出力となる
2. $x- = 1$ の場合
   上記の場合と同様に, B の前件が成立する. しかし, 発行された制約は A と矛盾せず, 極大無矛盾集合は{A, B}である.
3. $x- < 0 \vee x- > 2$ の場合
   B の前件は成立せず, A のみが採用される. ガードが成立せず, 後件は発行されない.

入力に対応した 3 通りの出力が考えられ, 表 4.1 で表される入出力の対応関係が導出できる. CalculateClosure では, 上記の結果を得るための計算を行う. なお, 入力に記号パラメタがあり, 記号パラメタに依って入力が複数の条件にまたがる場合は, 記号パラメタを分割して全てのケース実行する.

IP においては, ガード条件と無矛盾性が現在時刻から連続的に成立するかを判定する手続きが行われるが, 本質的には PP の場合と同じである.

4.1.4　最適化手法

本研究における MCS の最適化は, 表 4.1 のような入出力の対応表を実行時に作成する手法である. MCS に入力される制約が対応表の入力を満たしていれば, 対応する出力が MCS

の出力となる. そのような場合は入力条件との照合と代入で済ますことができ, 計算コストの高い MCS を短絡して出力を得ることができる. 同じ極大無矛盾集合が繰り返し採用されてフェーズが進行するようなモデルは MCS の入力に局所性があり, 本手法が有効に働く. 加えて, HyLaGI でモデリング対象のモデルにはそのような例が多いことが経験的に分かっている. このような手法は計算再利用 (Computation Reuse) [9] と呼ばれ, コンパイラの最適化手法等で用いられる.

4.2　提案手法の実行アルゴリズムと HyLaGI への実装

4.2.1　提案手法の実行アルゴリズム

本節では, 前節 4.1.4 で説明した最適化を組み込んだ HyLaGI の実行アルゴリズムの説明を行う. 図 4.4 は提案手法を組み込んだアルゴリズムの全体, 図 4.5, 図 4.6 は提案手法に必要な補助関数である.

以下の処理が元のアルゴリズムに追加されている.

1. CheckConsistencyPP および CheckConsistencyIP において, 通常通り制約の充足性判定をするのに加えて, 判定の有無に関わる制約を保存する.
2. MCS が結果を返した際, 上記の手続きで保存した制約の連言と出力の制約をペアにし, 入出力対応表のレコードとして保存. また, MCS の他の出力も関連付けて保存しておく. (図 4.4 における CP および CI が保存されるデータ構造になる)
3. MCS を呼び出す前に, 入出力対応表を確認して結果を再利用できるかどうかをチェックする. 再利用可能ならば, 出力と保存していた出力とシミュレーション進行に必要なデータを取り出し, MCS を呼び出さずにシミュレーションを進行する. (図 4.4 の 11 行目, 及び 19 行目)

1. について, 元のアルゴリズムの CheckConsistencyPP, CheckConsistencyIP では, 入力された制約が無矛盾であるための記号パラメタと矛盾するための記号パラメタのペアを返す. 提案手法では, その際に入力の制約から prev 変数を参照している制約から prev 変数以外の変数を除去し, 保存する. (図 4.5 の 6 行目) 抽出した制約は, CheckConsistencyPP, CheckConsistencyIP において, その入力に対して同様に矛盾/無矛盾であるという出力をするための条件である. これを今後 Path Constraint(PC) と呼ぶこととする. PC は, 直感的には手続き型プログラミングにおける条件文 (if 文) の条件部である. HyLaGI のような制約プログラミングにおいては矛盾/無矛盾であるための条件は明示

されないが, 入力に依って矛盾/無矛盾であるための条件を抽出することで, 無矛盾性判定の分岐に関わる入力条件を取り出している. 抽出した PC を満たす入力は, その PC を抽出した際の CheckConsistencyPP, CheckConsistencyIP と同じ判定結果 (矛盾/無矛盾) となる. CheckConsistencyIP では, 微分方程式を解くという点と, 無矛盾性判定をその時刻の正の近傍で行うという点で異なるが, 本質的には同じなので, 省略する.

2. について, 1. で保存した全ての PC の連言が入出力対応表の入力となる. 各 PC は, MCS で呼び出した CheckConsistencyPP, CheckConsistencyIP で同じ判定結果になるための条件なので, PC の連言は全ての CheckConsistencyPP, CheckConsistencyIP で同じ判定結果になるための条件である. 従って, これを満たすことは同じ極大無矛盾集合が導出されることを意味する.

4.2.2　HyLaGI への実装

意味論上は, HydLa の解軌道は prev 変数と過去に展開された □ 制約のみに因果的であるが, 文献 [7] で述べられているように, 現在の HyLaGI は前フェーズとの差分情報を保持して, 変化しない変数に関する制約の再計算を行っていない. そのため, 実装では前フェーズとの差分情報をキーとして, 各々のキーに対応した再利用表が保存される. 前のフェーズからの差分情報が同じならば、同じ再利用表を参照することとなる.

また, 小節において, MCS の入力が $T > 0$ であることと, 後件の □ 制約が展開されないことを仮定して議論を進めたため, 実装に当たってはそれを考慮する必要がある. つまり, 最初の PP においては入出力対応表の作成は行わず, 後件 □ が展開された場合は再利用表をリセットする.

4.3　実験による評価

公開されている HyLaGI リポジトリ [10] より性質の異なる例題を対象に, 評価実験を行った. 表 4.2 は実験環境である.

例題についての詳細を表 4.3 に示す

どれもフェーズ数を有界で打ち切らない限りシミュレーションが続く例題であり, 無矛盾性の判定処理が実行時間の多くを占める問題である. 実行フェーズを 300 で打ち切り, 最適化の実装前と後で実行時間の比較をした. 結果のグラフを図 4.7, 4.8, 4.9, 4.10 に示す.

グラフは, 最適化実装前と実装後の, 極大無矛盾集合導出処理と全実行時間における, 実行フェーズと実行時間の関係を表す. また, 実行後は再利用表の照合と代入についてのグラフも追加されている. また, 再利用表の作成にかかるコストは極大無矛盾集合のグラフに反映されている.

図 4.7 のプログラムでは, 温度が閾値に到達したら温度の微分方程式を変えることを繰り返す. PP, IP 共に, 2 パターンの極大無矛盾集合が採用されることを繰り返す. 3 フェーズまでは, 再利用表の作成コストが発生するため, 最適化前よりも余計なコストがかかるが, それ以降は以前と同じ極大無矛盾集合が採用されることを繰り返すため, 関数 `MCS` の

OS	Gentoo Linux (release 2.6)
CPU	AMD Ryzen 5 2600 1950X 16-Core Processor
Memory	16
Backend	Mathematica 12.0.0

表 4.2　実験環境

プログラム名	説明	特徴
thermostat	室温の調整	周期的な繰り返し
dose	定期的な投薬	解析の難しい数式が存在する
simple_water_tanks (3 tanks)	3 つのタンクの水量制御	無矛盾性判定のコストが大きい
simple_water_tanks (5 tanks)	5 つのタンクの水量制御	非周期的な動作かつ, 無矛盾性判定のコストが非常に大きい

表 4.3　実験対象のプログラム

処理を短絡し続ける. 結果, グラフで表される様に, 実行時間の多くを占めていた極大無矛盾集合の導出処理のボトルネックが解消されるようになった. 図 4.8 のプログラムは, 投薬を繰り返すハイブリッドシステムのモデルである. このプログラムの記号実行では, 解析的に解くのに時間のかかる計算式が存在し, 時間進行と共に複雑化していく数式が原因で実行時間が増大していく. 最適化により実行コストは削減できることが分かるが, 再利用表の照合と代入がボトルネックになっている. これは, 再利用表から出力を得た後に行う計算式の簡約が原因である. 最適化により極大無矛盾集合の導出において幾度も計算式を簡約する必要はなくなるが, 実質的に解析の難しい数式を簡約することは避けられず, ボトルネックの解消には至らなかった. 図 4.9, 図 4.10 のプログラムはそれぞれ 3 つ, 5 つのタンクの水量を調節するプログラムである. 特に 5 つのタンクがあるモデルは非周期的な動作を繰り返し, 極大無矛盾集合の求解コストも高い. グラフから分かるように, 再利用表の作成はなかなか収束しないが 75 フェーズあたりで収束し, それ以降は全て再利用表の照合と代入で済んでいる. 結果, 実行コストの殆どを占めていた極大無矛盾集合の導出コストが解消され, 実行時間の大幅な削減を実現した.

Require: HydLa プログラム *HydLa*,
　　最大時刻 *MaxT*

1: $MS := TopologicalSort(HydLa)$
2: $V := GetVariables(HydLa)$
3: $T := 0$
4: $S := true$
5: $P := true$
6: $E := \emptyset$
7: $CP := \emptyset$
8: $CI := \emptyset$
9: **while** $T < MaxT$ **do**
10:　$S := Subst(S, T)$
11:　$(S, P, \_, \_) := GetCacheResult(S, P, CP)$
12:　**if** $S = false$ **then**
13:　　$(S, P, E, CP, \_, \_) :=$
　　$MCS(S, MS, E, P, T, CP, CheckConsistencyPP)$
14:　**end if**
15:　**if** $S = false$ **then**
16:　　Break
17:　**end if**
18:　$(S, P) := AddParameters(S, P, V)$
19:　$(S, P, A_-, A_+) := GetCacheResult(S, P, CI)$
20:　**if** $S = false$ **then**
21:　　$(S, P, E, CI, A_-, A_+) :=$
　　$MCS(S, MS, E, P, T, CI, CheckConsistencyIP)$
22:　**end if**
23:　$S := SolveDifferentextitalEquatextiton(S)$
24:　**if** $S = false$ **then**
25:　　Break
26:　**end if**
27:　$(MinT, P) := GetElement(CompareMinTime($
　　$(\bigcup_{(g \Rightarrow c) \in A_-} FindMinTime(Subst(g, S), P))$
　　$\cup (\bigcup_{(g \Rightarrow c) \in A_+} FindMinTime(Subst(\neg g, S), P))$
　　$\cup \{(MaxT - T, true)\}))$
28:　$T := MinT + T$
29: **end while**

図 4.4　提案手法を組み込んだ HyLaGI の実行アルゴリズム

Require: 前フェーズの制約ストア S_{prev}
制約ストア S
記号定数の条件 P
Path Condition PC
Ensure: 充足の有無
記号定数の条件 S
新しい Path Condition

1: $V := GetVariables(S)$
2: $P_{tmp} := \exists V(S \wedge P)$
3: **if** $P_{tmp} = false$ **then**
4: 　**return** $(false, P, true)$
5: **end if**
6: $P_{tmp} := \exists V(S \wedge S_{prev} \wedge P)$
7: $newPC := removeUnnessesaryCondition(S)$
8: **if** $P_{tmp} = false$ **then**
9: 　**return** $(false, P, PC \wedge \neg newPC)$
10: **else if** $P_{tmp} = P$ **then**
11: 　**return** $(true, P, PC \wedge newPC)$
12: **else**
13: 　**return** $GetElement($
$\{(true, P_{tmp}, PC \wedge newPC),$
$(false, P \wedge \neg P_{tmp}, PC \wedge \neg newPC)\})$
14: **end if**

図 4.5　提案手法を組み込んだ checkConsistencyPP のアルゴリズム

Require: 前フェーズの制約ストア S_{prev}
　　記号定数の条件 P
　　入力条件と出力のリスト C
Ensure: 採用される制約
　　記号定数の条件
　　成立した条件付き制約
　　成立しない条件付き制約

1: $V := GetVariables(S)$
2: $P_{tmp} := false$
3: $R := \emptyset$
4: **for all** $(prevCond, S, A_+, A_-)$ in C **do**
5: 　$P_{true} := \exists V(prevCond \wedge S_{prev} \wedge P)$
6: 　**if** $P_{true} \neq false$ **then**
7: 　　$R := Append(R, (S, P_{true}, A_+, A_-))$
8: 　　$P_{tmp} := P_{tmp} \vee P_{true}$
9: 　**end if**
10: **end for**
11: **if** $P_{tmp} \neq P$ **then**
12: 　$R := Append(R, (false, P \wedge \neg P_{tmp}, \_, \_))$
13: **end if**
14: **return** $GetElement(R)$

図 4.6　GetCacheResult のアルゴリズム

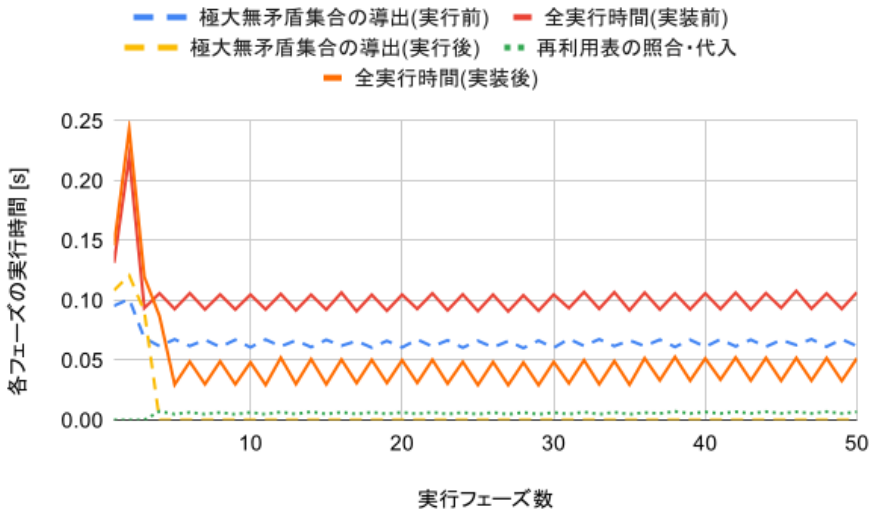

図 4.7　thermostat の実行フェーズと実行時間の関係の比較

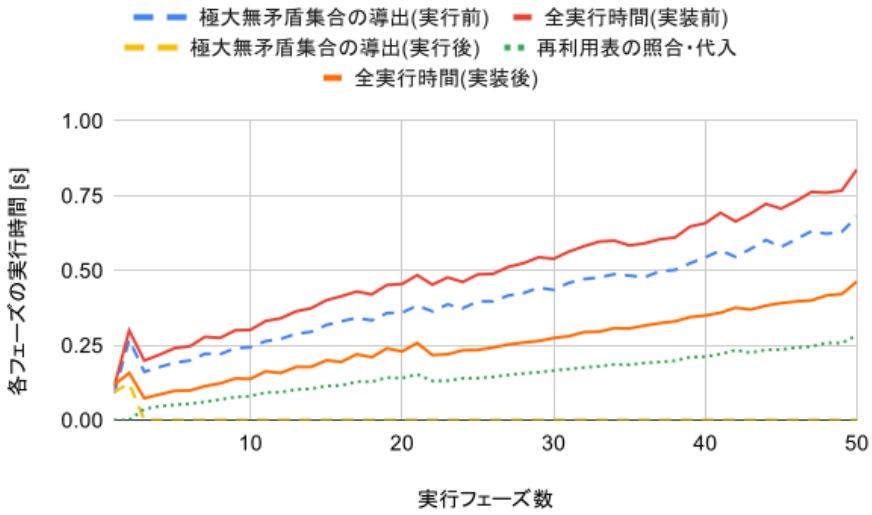

図 4.8　dose の実行フェーズと実行時間の関係の比較

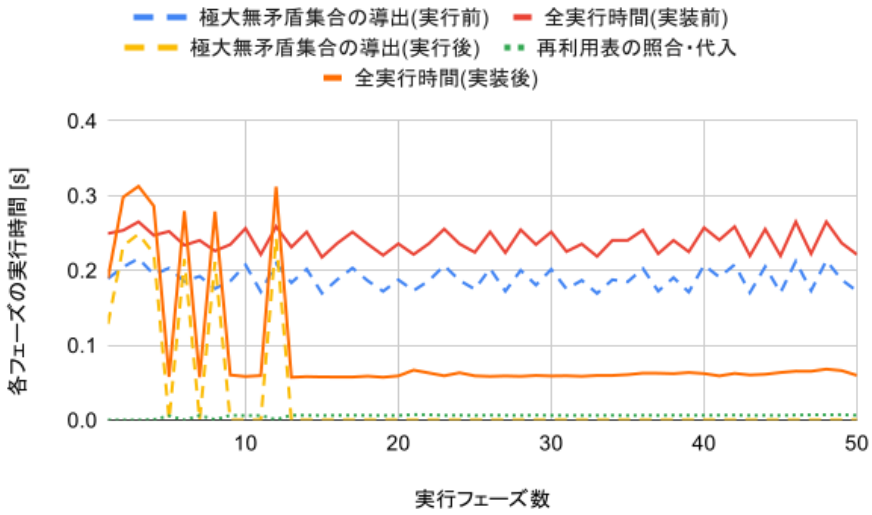

図 4.9　simple_water_tanks(3tanks) の実行フェーズと実行時間の関係の比較

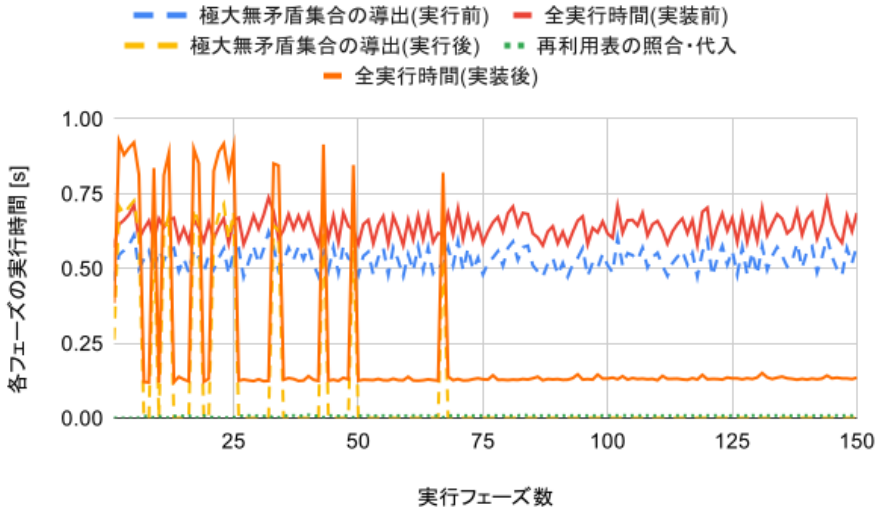

図 4.10　simple_water_tanks(5tanks) の実行フェーズと実行時間の関係の比較

第 5 章

分枝限定法を用いた離散変化時刻導出の最適化

ハイブリッドシステムのシミュレーションでは, 離散変化が発生するタイミングを導出する必要がある. HyLaGI において, 離散変化の要因が多数ある問題のシミュレーションにおいては, 離散変化の原因を全探索するが故に離散変化時刻の導出が実行のボトルネックになる場合がある. 本研究は, そのような問題を効率的に解くアルゴリズムを設計し, HyLaGI に実装することが目的である. 結論として, 離散変化時刻を導出する問題を部分問題に分割し, 分枝限定法を用いて効率的に解を求めるためのアルゴリズムを設計し, HyLaGI へ実装した.

5.1 離散変化時刻の導出手続きについて

HydLa においては, 条件付き制約の前件の成立の有無が変化することで離散変化が発生する. HyLaGI では, `IP` において, 時刻の関数である変数がガード条件を満たす (あるいは満たさなくなる) 最小の時刻を求めることで実現している. 具体的には図 2.2 の 21 行目の処理が対応する. 成立した/成立していない全てのガード条件について, 変数の式を各ガード条件の前件の元で最小化問題を解き, その中で最小の時刻を求めるものである. 最小化問題の求解は, mathematica の `Minimize` 関数を用いている.

調べるべきガード条件が多数ある場合, この処理が実行におけるボトルネックとなることがある. モチベーションとなる例題として, 図 5.1 を挙げる. 図 5.1 は階段を跳ねる質点をモデリングしたプログラムであり, 階段を表す条件付き制約が階段の段数に比例して存在する. 図 5.2 はその実行結果である. 階段の段数に比例して実行時間が増大し, 実行時間

```
1  // #define N 6
2
3  INIT_X(v) <=> (x = 0 & x'=v).
4  X_MOVE <=> [](x'' = 0).
5  INIT_Y(h) <=> (y = h & y' = 0).
6  FALL(g) <=> [](y'' = -g).
7  WALL(w) <=> [](x- = w & 0<=y-<=2*N => x' = -x'-).
8
9  BOUNCE_ON_STEP_HOR(cornerx) <=>
10   []( (y- = N - cornerx-) & (cornerx- <= x- < cornerx- + 1)
11     => (y' = -9/10 * y'-) ).
12 BOUNCE_ON_STEP_VER(cornerx) <=>
13   []( (x- = cornerx-) & (N - cornerx- < y- <= N - cornerx- + 1)
14     => (x' = -x'-) ).
15 BOUNCE_HOR := { BOUNCE_ON_STEP_HOR(i) | i in {0..N} }.
16 BOUNCE_VER := { BOUNCE_ON_STEP_VER(i) | i in {0..N} }.
17
18 INIT_X(1), INIT_Y(N + 3),
19 (FALL(9.8) << BOUNCE_HOR),
20 (X_MOVE << BOUNCE_VER << WALL(0)).
```

図 5.1　階段を跳ねる質点の HydLa プログラム

の殆どを占めていることが分かる.

5.2　最適化に用いる手法

図 5.1 の効率の悪い点は, 直感的に明らかに確認不要な条件も含めて最小化時刻の探索をしている点である. 例えば, 質点が階段の前方を跳ねている場合は後方の探索は不要であることが多く, 通り過ぎた階段は質点の移動方向が変わらない限り探索の対象にするべきではない. 一般的に当たり判定を考える際は空間分割をして探索対象を限定するように, 探索対象を限定するための手法, アルゴリズムが必要である.

5.2.1　分枝限定法

分枝限定法 [11] は, 離散最適化を効率良く解くための汎用アルゴリズムである. 幾つかの制約条件の元で目的関数を最小化する問題において, 制約条件を部分問題に分割し, 部分問題の上界/下界を用いて最適値に寄与しない制約を枝刈りする手法である. ナップサック問題や巡回セールスマン問題等の離散最適化問題に有効な手法で, 一部の SAT ソルバに実

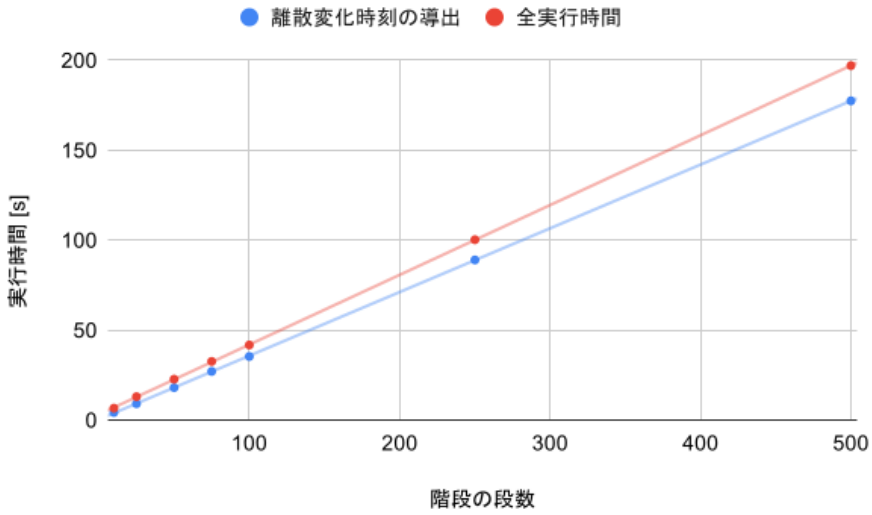

図 5.2　階段を跳ねる質点のプログラムにおける段数と実行時間の関係

装されている. なお本章では, 制約を集合として扱い, 集合の記法を用いて議論を進める点に注意して欲しい.

一般的な分枝限定法について説明する. 分枝限定法では, 分枝操作と限定操作を繰り返して最適値の探索を行う. 分枝操作では, 目的関数にかかる制約条件を幾つかの制約条件に分割し, 元の問題を複数の部分問題に分割する. 例えば, $x(t)$ を制約 G の元で最小化する問題において分枝操作を適用することを考える. 制約 G を $G = G_1 \cup G_2 \cup \cdots \cup G_n$ を満たすような $G_1 \ldots G_n$ に分割し, 各々のもとで目的関数を最小化する問題を解く. それらの解 $z_1 \ldots z_n$ の内, 最も小さいもの, すなわち $min(z_1, \ldots, z_n)$ が元の問題の解である.

分枝操作だけでは問題を分割しただけであり, 枝刈りには寄与しない. 部分問題をできるだけ解かずに解を求めるために必要なのが限定操作である. 限定操作では, 各部分問題の制約を包含する別の制約に置き換え, 代わりにその制約の元で最適化問題を解く. 例えば, $x(t)$ を制約 G_i の元で最小化する際の限定操作は, 制約 $G_i \Rightarrow H_i$ を満たす H_i を作成し, $x(t)$ を制約 H_i の元で解くことで実現できる. 包含する制約で置き換えた問題を緩和問題といい, 緩和問題において以下の定理が成り立つ. 定理 1 では, 緩和問題を解くことで元の問題の最適値の下界を求めることができる. 部分問題の最適値の下界は探索における枝刈りに用いることが出来, 探索においてその下界の値よりも小さな値が解候補として求められた場合, その部分問題は最適値になりえないため, その部分問題の探索を打ち切ることができる.

分枝操作と限定操作を再帰的に繰り返すことで, できるだけ問題をブレークダウンせず

に効率良く解を探索する.

定理 1. 制約条件 $\boldsymbol{x} \in X_p$ 下で $f(\boldsymbol{x})$ を最小化する問題 P と, 制約条件 $\boldsymbol{x} \in X_{rp}$ 下で $f(\boldsymbol{x})$ を最小化する問題 RP において,
$X_p \subset X_{rp}$ が成立するならば, P の最適値 z_p* と RP の最適値 $z_{rp}*$ について, $z_p* \geq z_{rp}*$.

証明. $f(x_p*) = z_p*$ を満たす P の最適解 x_p* について,
$x_p* \in X_p \subset X_{rp}$ より, x_p* は RP の実行可能解.
$\therefore z_p* \geq z_{rp}$ □

5.3　HyLaGI の離散変化時刻導出手続きへの分枝限定法の適用

HydLa における離散変化時刻の導出は, 時間の関数である変数をガード条件の元で最小化する問題である. 形式的には, 変数における等式制約 $S \Leftrightarrow x_1 = f_1(t) \wedge x_2 = f_2(t) \wedge \cdots \wedge x_n = f_n(t)$ とそれらを参照するガード条件の集合 $G = (g_1, g_2, \ldots, g_m)$ を用いて, HyLaGI の離散変化時刻の導出問題は以下の最小化問題として記述できる.

$$\text{Minimize } t \text{ such that } S \wedge (g_1 \vee g_2 \vee \cdots \vee g_m)$$

HydLaGI の実装では, 論理和を分解し一つ一つの制約に対して最小化問題の求解を繰り返し, 結果を比較する. 論理和のままソルバに解かせることも可能だが, 問題によっては (特に, パラメタがある場合) ソルバが停止したり求解に大きく時間がかかったりする. 極大無矛盾集合の導出もそうであるが, HyLaGI の実装では基本的にソルバの負担を考慮してできるだけ事前に小さな問題に分割してからソルバに解かせている.

5.3.1　汎用アルゴリズムの提案

HyLaGI の離散変化時刻の導出に分枝限定法を適用したアルゴリズムを図 5.3 に示す. この関数は, ガード条件の集合と変数の時間変化の等式, 記号パラメータを受け取り, 記号パラメータに対応する離散変化時刻を返す. 記号パラメータに対応して離散変化原因が異なる可能性があるため, 非決定性が存在する.

アルゴリズム全体では, 元の問題を幾つかの部分問題に分割し, 部分問題の解の下界で枝切りをしながら元の問題の解の探索を行う. 9 行目から 29 行目のループでは, プライオリティキューを用いた探索を行う. プライオリティキューの要素は探索中の部分問

題に対応しており, その要素は<部分問題の最適値の下界, 部分問題の制約条件, 部分問題の記号パラメータ, 探索完了フラグ>である. キューの優先度は, 部分問題の最適値の下界である. ループ内ではまず, 探索中のガード条件を包含する新しい制約を作成する (19 行目 `CalculateRelaxedGuards` 関数). これは, 入力 $(g_1, g_2, \ldots, g_n) = G_{curt}$ と出力 $(h_1, h_2, \ldots, h_m) = H$ について, $\bigvee_{i=1}^{n} g_i \Rightarrow \bigvee_{j=1}^{m} h_j$ が成立するような H を作成する. この操作の目的は, ガード条件を幾つかの制約に分割する分枝操作と, 分割後の各制約を包含する制約を作ることで緩和問題を作成する限定操作を行うことである. 探索対象の多数のガード条件をひとまとめにした緩和問題を作成することが目的のため, n より m がずっと小さいことが望ましい. 21 行目では作成した各緩和問題を解き, 解の下界を求める. 23 行目では, 解の下界が元の問題の実行可能解であるかを判定している. 導出した解の部分問題に所属するガード条件を満たすかをチェックし, 満たされていたらそれが部分問題の最適解である. 満たされているならばその部分問題のこれ以上の探索は不要なので, 探索完了フラグを True にしてプライオリティキューに詰む. 満たされてないならば, その部分問題の最適値は判明していないため, 求めた最適値の下界を更新してプライオリティキューに詰む. ループにおいて探索完了フラグが True の要素を pop した時, その記号パラメータに対応した最適値が求まったことになる (14 行目). 16 行目において, 最適値が求まった記号パラメータの探索を打ち切っている.

5.3.2　実装した具体的なアルゴリズム

アルゴリズム 5.3 は, 分枝限定法を用いた一般的なアルゴリズムである. 実装において具体化する必要があるのは, `CalculateRelaxedGuards` 関数である. この関数は, ガード条件をグルーピングし緩和問題を作成する分枝限定法の肝であるが, アルゴリズムではその手法について言及していない. 一般的に制約を緩和する方法は無数にあり, あらゆる問題で有効に機能する分枝操作, 限定操作を設計するのは現実的でない. 緩和問題として望ましい要件は以下の通りである.

- 元の問題よりも解きやすい (短い時間で解ける)
- 最適値と最適値の下界の差が小さい
- 緩和問題の解が許容解に含まれやすい

1 つ目は必須の要件であり, 緩和問題が元の問題より難しいと, そもそも部分問題に分ける必要がなく, 本末転倒である. 2, 3 つ目の性質は部分問題の収束のし安さに影響する. 最適値とその下界の差が大きいと, 良い解を持っている可能性が否定できないため, 枝刈りされ

にくくなる. また, 緩和問題の解が許容解に含まれていると, 下界=部分問題の最適値となるため, 部分問題の探索を打ち切ることができる. 本研究では, 動機である二次元の衝突問題を記述した HydLa プログラムで有効に機能するアルゴリズムを設計, 実装した. アルゴリズムの適用対象として, ガード条件がある 2 つの変数のどちらかまたは両方を参照した一次方程式, 一次不等式であることを仮定した. `CalculateRelaxedGuards` 関数の実装は以下の通りである.

1. 分枝操作としては, 変数の式 $x = f(t)$ における $x = f(0)$ で x-y 座標平面を分割し, それぞれに包含されるガード条件を同じグループとする. ガード条件が境界面をまたぐ場合は, どちらかに所属させるか独立させる.
2. 限定操作としては, 同じグループに所属するガード条件を凸包で包含することで実現する.

緩和問題として凸包は, 前述の緩和問題として望ましい要件と照合すると, 以下のようになる.

1. 複数のガード条件に対して最小化問題の求解するよりも、一つの凸包下で最小化問題の求解を行うことの方が実行コストが小さいと考えた
2. 最適値と最適値の下界の差が小さいかは, 実装に依る
3. 凸包の表面とガード条件が接する線分が最適値に寄与する制約の場合, 緩和問題の解が許容解になる

図 5.4 に, 上記の手続きを具体化してアルゴリズムを実装した際の, プログラム 5.1 における探索の様子を示す. 図示の簡単化のため, 質点の初期位置を数フェーズ後のものに変更している.

まず, `CalculateRelaxedGuards` 関数の実装が (2), (3) である. (2) では, 質点の x 座標で空間を 2 分し, 同じ空間に含まれる線分を同じグループとする. 次に, 同じグループに属する線分を凸包で包含する. (3) では, 空間をまたがった線分をどちらかのグループに属させる. この時に重要なのが, 質点の初期位置を凸包に包含されないようにする点である. (4) で作成した凸法のもとで最小化問題を解くが, 質点の初期位置が凸法に包含されていると最適値 (部分問題の下界) が $t = 0$ となり, 意味のない探索が発生してしまうからである. 従って, 凸包と点の包含判定 (初期値が区間の場合は交差判定) を行いつつ, 可能ならば線分をどちらかのグループにマージする. これで出来た凸包が分枝限定法における制約が緩和された部分問題になる. 結果, 左右の凸包と, 質点が乗っている, マージできなかった線分の 3 つの部分問題がある状態である. (4) では, 質点の解軌道と凸包を表す制約下で最小

化問題を解くことで, 各部分問題の最適値の下界を求める. 左の凸包と線分では, 最小化問題の最適値が無い (あるいは, $t = \infty$ の下界を持つ) ため, その部分問題の探索を打ち切って良い. これは, 直感的には, 質点が凸包に衝突しなければ, 凸法に内包される全ての線分に衝突しないことを意味し, 定理 1 より明らかに示される性質である. 右の凸包の元で最小化問題を解くと最適値が導出でき, それがその部分問題の下界となる. この時, 導出された最適値が元の問題の実行可能界であるかの判定を行う. これは, 直感的には質点が階段の段に触れているかどうかを判定すればよく, 実際の処理では導出された時刻を質点の解軌道の式に代入して, ガード条件を満たすかの判定を行う. 結果, 満たされていないため, この部分問題はまだ解けておらず, 探索を進める必要がある. (5), (6) では, 右の凸包に対応する部分問題の探索を続ける. この後の処理は, (2)-(4) と同様の操作を再帰的に繰り返す. (6) では, 質点が階段の段に接触しており, 部分問題の下界が実行可能解であると判定されるため, 元の問題の暫定的な最適値を更新して探索を終了する. これで全ての部分問題の探索が終了するため, 暫定的な最適値が元の問題の最も良い解となり, 離散変化時刻が求まる.

5.4　実装と例題を用いた実験

実装した手法を, 階段を跳ねる質点のプログラム (図 5.1) と複数の壁のある部屋で衝突を繰り返す質点のプログラム (図 5.5) を対象に評価した. 実行結果のグラフを図 5.6, 5.7 に示す.

図 5.6 より、離散変化時刻導出のボトルネックが解消されていることが分かる。アルゴリズムは図 5.4 のように実行される。階段の段数が増加しても全ての段に離散変化時刻の導出をすることはせず、凸包で覆われた複数の段がまとめて探索対象から外されたり、探索前に解が見つかり探索が打ち切られたりすることで、最小化問題の求解の回数が抑えられる。ここで、提案アルゴリズムの実行時間の詳細を図 5.8 に示す。

図 5.8 では、最適化後のアルゴリズムで実行した階段プログラムの、N=100, N=500 における、1 フェーズ目, 20 フェーズ目の実行時間の詳細である。各フィールドは、アルゴリズムの関数に対応する。図より、最小化問題の求解である `FindMinTime` の実行コストは階段の段数 N に依らないことがわかる。最適化前では `FindMinTime` の実行時間が N に比例して増加していることから、FindMinTime の実行回数を N に依らなくしたことが実行コストの削減に寄与していると考察できる。なお、20 フェーズ目で `FindMinTime`, `CheckGuardSat` の実行時間が増加しているのは、フェーズ進行に伴って数式が複雑化し、記号計算が難しくなったためであると考えられる。`FindMinTime` の実行時間が N に依らなくなった代わりに、`CalculateRelaxedGuards` のコストが N に依って増加している。

`CalculateRelaxedGuards` では凸包の作成を行っており、凸包の作成は頂点の数 n に対して計算量が $O(nlogn)$ であるため、この結果は自然である。図 5.6 より、凸包作成にかかるコストは最小化問題の求解コストと比較して小さいことが分かる。

図 5.7 は、複数の障害物のなかで衝突を繰り返す質点である。本プログラムでも離散変化の対象が多く、その殆どが考慮するべきでない条件であるが、最適化によって探索対象から除外でき、実行コストの削減が実現できた。

Require:

1: G : ガード条件
2: S : 制約条件
3: P : 記号定数の条件

Ensure:

4: 最小離散変化時刻
5: 記号定数の条件
6: $PQ := PriorityQueue()$
7: $Sol := \{\}$
8: $PQ.push(\langle 0, G, P, false\rangle)$
9: **repeat**
10: $\langle T_{curt}, G_{curt}, P_{curt}, tf_{curt}\rangle := PQ.pop()$
11: **if** $P_{curt} = false$ **then**
12: *continue*
13: **end if**
14: **if** $tf_{curt} = true$ **then**
15: $Sol := Sol \cup \{\langle T_{curt}, P_{curt}\rangle\}$
16: $PQ := \bigcup_{\langle T,g,p,tf\rangle \in PQ}\{\langle T, g, \neg P_{curt} \wedge p, tf\rangle\}$
17: *continue*
18: **end if**
19: $H := CalculateRelaxedGuards(G_{curt})$
20: **for** $h \in H$ **do**
21: $MinResult := FindMinTime(Subst(h, S), P_{curt})$
22: **for** $\langle T_{min}, p\rangle \in MinResult$ **do**
23: $CGSResult :=$
$CheckGuardSat(Subst(S, t = T_{min}), G_{curt}, p)$
24: **for** $\langle tf, p\rangle \in CGSResult$ **do**
25: $PQ.push($
$\langle T_{min}, \{g | g \in G_{curt} \wedge g \Rightarrow h\}, p, tf)\rangle$
26: **end for**
27: **end for**
28: **end for**
29: **until** $PQ = \{\}$
30:
31: **return** $GetElement(Sol)$

図 5.3　分枝限定法を用いた離散変化時刻導出問題の非決定アルゴリズム

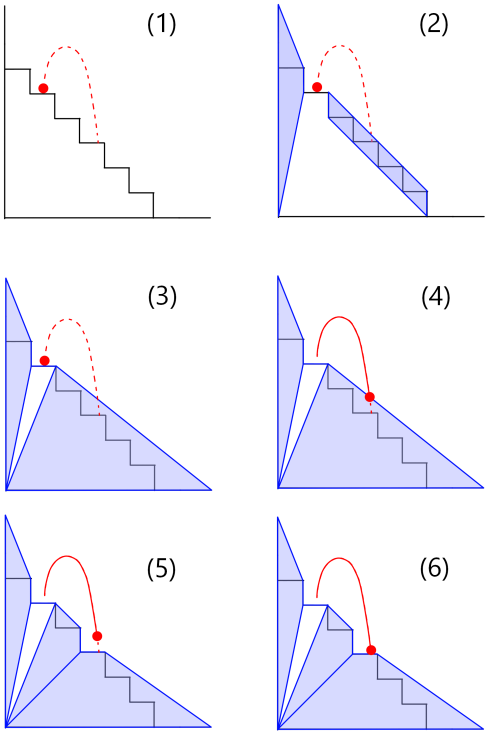

図 5.4　階段を跳ねる質点のプログラムに提案アルゴリズムを適用した解探索の流れ

```
1  // #define N 100
2  // #define M 50
3
4  INIT <=> x=0.5 & y=0 & x'=1 & y'=1.
5  CONSTX <=> [](x''=0).
6  CONSTY <=> [](y''=0).
7  XWALL <=> [](x-=0 & 0<=y-<=N => x'=-x'-).
8  XWALL2 <=> [](x-=N & 0<=y-<=10 => x'=-x'-).
9  YWALL <=> [](1<=x-<=N & y-=0 => y'=-y'-).
10 YWALL2 <=> [](0<=x-<=N-1 & y-=N => y'=-y'-).
11 BOUNCE_ON_STEP_VER1(cornerx) <=> [](  (x- = cornerx-) & (0 <= y- <= N - 1) => (x
       ' = -x'-) ).
12 BOUNCE_ON_STEP_VER2(cornerx) <=> [](  (x- = cornerx-) & (1 <= y- <= N) => (x' =
       -x'-) ).
13 BOUNCE_VER1 := { BOUNCE_ON_STEP_VER1(i*2-1) | i in {1..M-1} }.
14 BOUNCE_VER2 := { BOUNCE_ON_STEP_VER2(i*2) | i in {1..M-1} }.
15
16 INIT.
17 CONSTX << (XWALL, XWALL2, BOUNCE_VER1, BOUNCE_VER2).
18 CONSTY << (YWALL, YWALL2).
```

図 5.5　複数の壁のある部屋で衝突を繰り返す質点のプログラム

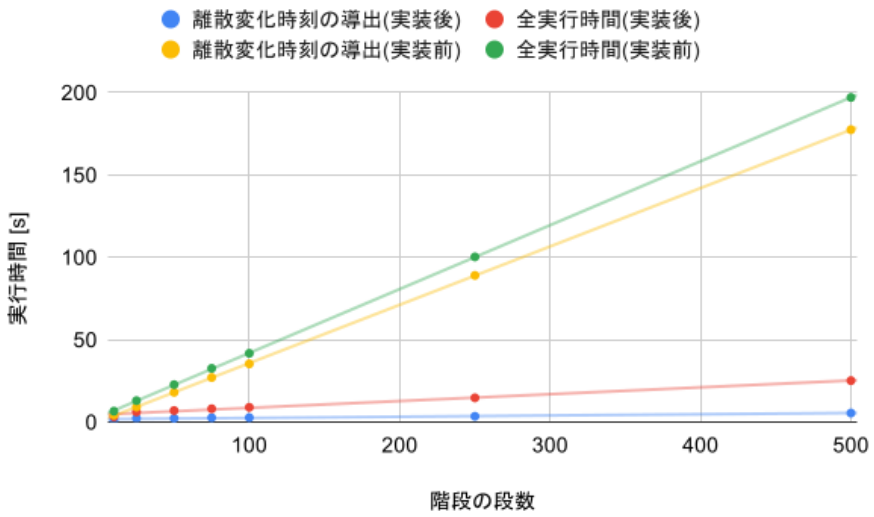

図 5.6　プログラム 5.1 における最適化前後の, 階段の段数と実行時間の関係

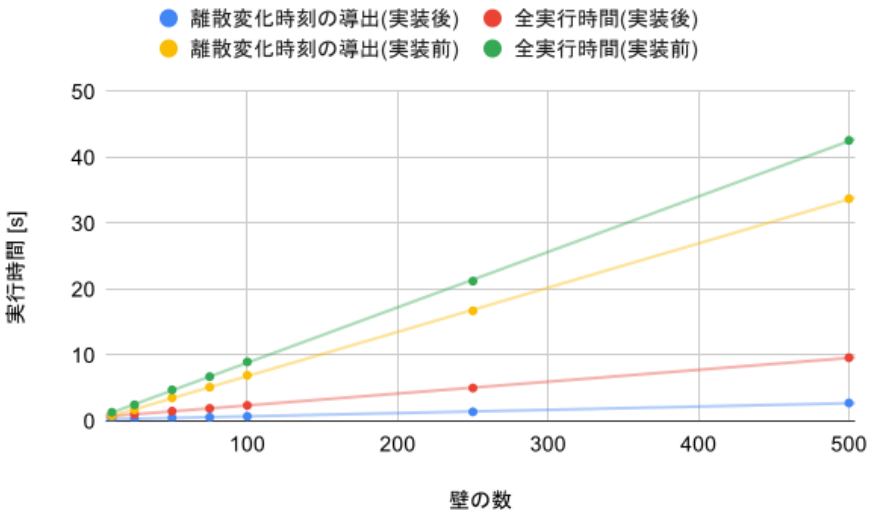

図 5.7　プログラム 5.5 における最適化前後の, 壁の数と実行時間の関係

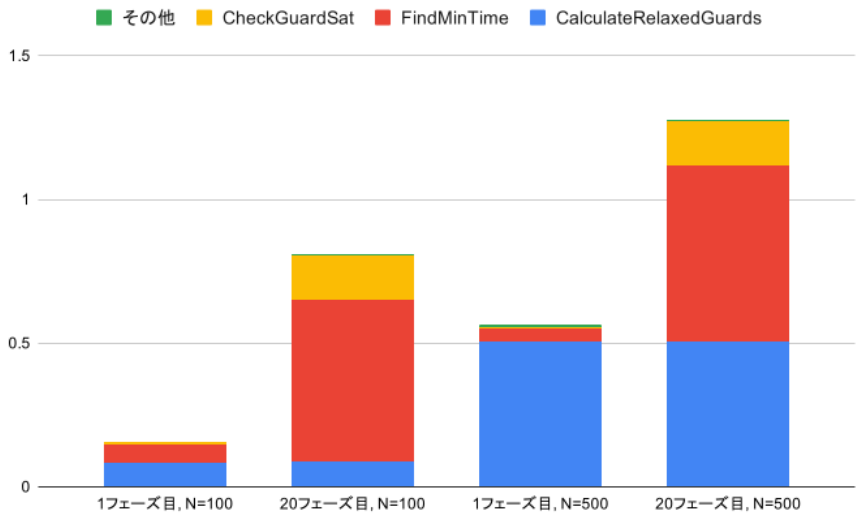

図 5.8　プログラム 5.1 における、最適化後の実行時間の詳細

第 6 章

関連研究

6.1 Acumen

Acumen [13] はハイブリッドシステムを記述, シミュレートする手続き型言語である. バックエンドに複数のソルバを備えており, 微分方程式を Runge-Kutta 法や Taylor 展開により求解する数値計算モードと区間演算による精度保証モードがある. その他, 軌道の IDE や 3D 描画といった豊富な機能を持つ. HyLaGI とは不定値を扱える点と精度を保証したシミュレーションが可能な点が同じであるが, 手続き的な構文である点と方程式や微分方程式の求解を解析的に行わないという点で異なる.

6.2 Flow*

Flow* [12] は, 非線形なハイブリッドシステムの到達可能性を検証するツールである. テイラーモデル法により精度保証シミュレーションを可能とするのが特徴である. Flow* は離散変化の発生を domain conclusion によって導出しているが, その手続きの最適化として分枝限定法を用いている. 本研究では離散変化の発生を記号的に導出する際の対象を限定するために分枝限定法を用いており, その点で異なっている.

6.3 HyLaGI の最適化の先行研究

HyLaGI の最適化には以下の先行研究 [7] がある

- 前フェーズとの差分のみを計算し、再計算を避ける最適化
- 動的な極大無矛盾集合の導出による最適化

前者の最適化は変化していない変数の再計算を避ける手法で, 本研究の計算再利用は過去の全てのフェーズの計算結果を覚えておいて再利用する手法であり, 異なる着眼点である.

第 7 章

まとめと今後の課題

7.1　まとめ

本研究では HyLaGI のスケーラビリティ向上を目的とし, 記述については存在量化子の実装を, 処理系においては計算の再利用による最適化と最小離散変換時刻の導出処理の最適化を行った. 存在量化子は変数をモジュールから漏れ出すことを防ぎ, 複雑なモデルもモジュールの合成で構成することを促進するものである. また, モジュールの再帰的な定義を可能にしたので, 言語としての表現能力が向上した. 最適化においては, どちらも HyLaGI の要である記号処理のコストを改善するものであり, 問題に依ってはボトルネックを解消することができた. 計算の再利用は, 過去と同じ極大無矛盾集合が繰り返し採用される問題の計算コストを削減する最適化であり, その極大無矛盾集合が採用される条件と出力を保存し, 同じ計算が繰り返されるのを防ぐものである. 結果として, 極大無矛盾集合の判定処理がボトルネックとなる問題の実行時間を 50% 以上削減することができた. 最小離散変化時刻の最適化は, 離散変化の原因となるガード条件の探索に分枝限定法を導入した汎用的な非決定アルゴリズムの提案と, 2 次元の衝突問題を対象をして実装を行った. その結果, 動機となる例題のボトルネックを解消することができ, 実行時間の大幅な短縮を実現した.

7.2　今後の課題

本研究では分枝限定法を用いた離散変化時刻の探索の汎用アルゴリズムを設計したが, 具体的な実装は適用するための制約が多く, 一部の例題にしか適用できない. 分枝限定法自体も汎用アルゴリズムであるが, うまく機能させるためには問題のドメインごとに適切な緩和問題を考察する必要がある. さらなる拡張のためには, モチベーションとなる問題

を考察してアルゴリズムにおける `CalculateRelaxedGuards` 関数を問題の特性に応じて実装する必要があると考えている.

また, HyLaGI に依然として残る課題であるが, フェーズの進行に伴って数式が肥大化した場合, 実行時間が増加したり, 求解に失敗したりする. このような場合, 数式を区間値で包含して複雑な数式を安全に簡単化するという手法が考えられるが, 十分な考察が成されていない.

謝辞

本研究にあたって、多くの方々にお世話になりました。上田和紀教授には、日頃の研究活動にて熱心なご指導を頂いたのに加えて、合宿や学会出張においても大変お世話になりました。深く感謝いたします。また、研究室の皆様、研究生活において仲良くしていただきありがとうございます。HyLaGI のデバッグで共に苦しんだり蒙古タンメン中本に連れて行ったりしてくれた渋井くんに感謝いたします。また中本行きましょう。学会の〆切で共に苦しんだりジムに行ったりしてくれた齋藤くんに感謝いたします。シェル芸で天下を取る日を楽しみにしております。なんだかんだバイトで色々やらせてしまった角谷くんに感謝いたします。なんとか卒業&就職できるのを切に (切に!) 願っております。恒川さんには、Halmstad の出張や日頃の研究室での生活等でお世話になりました。博士課程も頑張ってください。山田くんは班の同期がおらず色々と大変かと思いますが、卒業まで頑張ってください。HyLaGI で困ったことがあったら遠慮なくきいてください。最後に、常に支えてくださった家族に深く感謝いたします。

参考文献

[1] Lunze, J : Handbook of Hybrid Systems Control: Theory, Tools, Applications, Cambridge University Press, 2009.

[2] Borning, A., Freeman-Benson, B. and Wilson, M.: Constraint Hierarchies, Lisp and Symbolic Computation, Vol. 5, No. 3, 1992, pp. 223–270.

[3] 上田和紀, 石井大輔, 細部博史 : ハイブリッド制約言語 HydLa の宣言的意味論, コンピュータソフトウェア, Vol. 28 No. 1, 2011, pp. 306–311.

[4] 上田和紀, 石井大輔, 細部博史: 制約概念に基づくハイブリッドシステムモデリング言語 HydLa, SSV2008(第 5 回システム検証の科学技術シンポジウム), 2008.

[5] 松本翔太: Validated Simulation of Parametric Hybrid Systems Based on Constraints, 早稲田大学大学院基幹理工学研究科, 博士論文, 2017.

[6] 廣瀬賢一, 大谷順司, 石井大輔, 細部博史, 上田和紀: 制約階層によるハイブリッドシステムのモデリング手法, 日本ソフトウェア科学会第 26 回大会, 2D-2, 2009.

[7] 河野文彦, 小林輝哉, 松本翔太, 上田和紀: 数式処理に基づくハイブリッドシステムシミュレータ Hyrose の大規模モデルシミュレーションに向けた拡張，日本ソフトウェア科学会第 31 回大会, 2014.

[8] Wolfram Research, Inc., Mathematica, http://www.wolfram.com/mathematica/index.html.

[9] Ding, Y., Li, Z. : A Compiler Scheme for Rusing Intermediate Computation Results, Proceedings of the International Symposium on Code Generation and Optimization, 2004, pp. 277–288.

[10] HyLaGI examples, https://github.com/HydLa/HyLaGI/tree/master/examples.

[11] Land, Ailsa H., and Alison G. Doig. An automatic method of solving discrete programming problems. Econometrica: Journal of the Econometric Society (1960): 497-520.

[12] Chen, Xin, Erika Abraham, and Sriram Sankaranarayanan. "Taylor model flowpipe construction for non-linear hybrid systems." 2012 IEEE 33rd Real-Time Systems Symposium. IEEE, 2012.

[13] Duracz, Adam. Rigorous simulation: its theory and applications. Diss. Halmstad University Press, 2016.

発表論文

[1] 佐藤柾史, 上田和紀: ハイブリッドシステムモデリング言語 HydLa における変数と制約階層の動的生成記法 人工知能学会全国大会 2018.

[2] 佐藤柾史, 上田和紀: 実行パスの動的解析によるハイブリッドシステム処理系 HyLaGI の最適化 日本ソフトウェア科学会全国大会, 2018.

[3] 佐藤柾史, 上田和紀: ハイブリッド制約処理系 HyLaGI における分枝限定法を用いた離散変化時刻導出手法 第 21 回プログラミングおよびプログラミング言語ワークショップ (ポスター発表), 2019

[4] 佐藤柾史, 上田和紀: ハイブリッド制約処理系 HyLaGI における分枝限定法を用いた離散変化時刻導出手法 人工知能学会全国大会 2019.

[5] 山田悠之介, 佐藤柾史, 上田和紀: Constraint-based Modeling and Symbolic Simulation of Hybrid Systems with HydLa and HyLaGI, CyPhy2019, 2019.